View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University
Purdue e-Pubs

ECE Technical Reports | Electrical and Computer Engineering

9-20-1994

Scheduling a Superscalar Pipelined Processor Without Hardware Interlocks

Heng-Yi Chao
Purdue University School of Electrical Engineering

Mary P. Harper
Purdue University School of Electrical Engineering

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Chao, Heng-Yi and Harper, Mary P., "Scheduling a Superscalar Pipelined Processor Without Hardware Interlocks" (1994). *ECE Technical Reports*. Paper 198.
http://docs.lib.purdue.edu/ecetr/198

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

SCHEDULING A SUPERSCALAR PROCESSOR WITHOUT HARDWARE INTERLOCKS

HENG-YI CHAO
MARY HARPER

TR-EE 94-29
SEPTEMBER 1994

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Scheduling a Superscalar Pipelined Processor Without Hardware Interlocks

Heng-Yi Chao and Mary P. Harper
School of Electrical Engineering
128.5 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285
email : *hengyi@ecn.purdue.edu, harper@ecn.purdue.edu*

September 20, 1994

Abstract

In this paper, we consider the problem of scheduling a set of instructions on a single processor with multiple pipelined functional units. In a superscalar processor, the hardware can issue multiple instructions every cycle, providing a fine-grained parallelism for achieving order-of-magnitude speed-ups. It is well known that the problem of scheduling a pipelined processor with uniform latencies, which is a subclass of the problem we consider here, belongs to the class of NP-Complete problems. We present an efficient lower bound algorithm that computes a tight lower bound on the length of an optimal schedule, and a new heuristic scheduling algorithm to provide a near optimal solution. The analysis of our lower bound computation reveals that if a task matches the hardware or the type of instructions is uniformly distributed, then issuing five instructions per cycle can achieve a speed-up; however, if the task is a bad match with the hardware, then issuing more than three instructions per cycle does not provide any speed-up. The simulation data shows that our lower bound is often very close to the solution obtained by our heuristic algorithm.

key words: superscalar, pipeline scheduling, VLIW, lower bound.

Contents

1 Introduction

To exploit the fine-grained parallelism in programs, two approaches have been used, the hardware approach and the software approach. The MIPS processor [22, 26] and the VLIW architecture [13, 19, 20] represent the software approach, in which the compiler has the entire responsibility for the correct execution of the compiled code. For VLIW processors, in each instruction word, a *field* is reserved for each functional unit which controls the behavior of the corresponding functional unit. On the other hand, superscalar processors [5, 10, 11, 17, 29, 31, 33, 34] represent the hardware approach, where the correct execution of programs relies on the pipeline interlocks or conflict management hardware.

For VLIW processors, the scheduling is done at compile time; while for supel-scalar processors, the scheduling is done at run time. Because there are no hardware interlocks, the hardware design of VLIW machines is simpler and faster. However, the potential drawbacks of this approach include the possible waste of memory due to long instructions and the need for high memory bandwidth. In a VLIW processor, many functional units may remain idle because of the dependencies among instruction:;. The code density problem is solved by using a variable-length representation in main memory at the cost of an extra mechanism to expand the compacted code into the cache [13]. The VLIW design suggests that the hardware and software must work closely to achieve a higher performance.

The superscalar pipelined design has become popular for many new generation processors [5, 11, 29, 31, 33]. In a superscalar pipelined processor (SPP), multiple instructions are fetched and decoded during each cycle, and there are multiple pipelined functional units that can execute these instructions concurrently. For example, the IBM RS/6000 processor [5, 31] has a four-word instruction fetch bus and can execute as many as four instructions (a branch, a condition-register instruction, a fixed-point instruction, and a floating-point instruction) in a single cycle. The Pentium processor [33] can fetch and decode two instructions at a time. It has two integer ALUs and a pipelined floating-point unit that consists of a multiplier, an adder, and a divider. The Motorola 68060 processor [11] has a four-stage instruction fetch pipeline, dual four-stage operand execution pipelines, and a floating-point unit that consists of a multiplier, an adder, and a divider.

The SPP scheduling problem involves determining a minimum length schedule for a set of instructions on a superscalar pipelined processor. Each instruction must be executed on a pipeline of the same type (pipeline and functional unit are used interchangeably in this paper). Each

functional unit is pipelined with a possibly different number of stages for execution. For example, the latencies for a floating-point addition, multiplication, and division in a Motorola 68060 processor requires 3, 4, 24 cycles, respectively [11]. The goal of an SPP scheduling algorithm is to determine a minimum length schedule by reordering instructions and inserting necessary no-ops (or stalls) such that the compiled code is guaranteed to contain no pipeline hazards. For an SPP scheduling problem instance **I**, let $S^*(I)$ be the optimal solution and $S^A(I)$ be the solution obtained by algorithm A. In this paper, if a quantity implicitly depends on I, then **I** is dropped from the notation.

It is well known that the problem of scheduling a pipelined processor with uniform latencies, which is a subclass of the problem we consider here, belongs to the class of NP-Complete problems [7, 22, 30]. For NP-complete problems, it may not be possible to find optimal solutions in polynomial time. However, efficient approximation algorithms exist for many of these problems. The quality of an approximation algorithm A is often measured by its guaranteed worst-case performance ratio $R(A)$ [21]. Comparing two algorithms solely using $R(A)$ bounds can be misleading because the average-case performance may differ significantly from the worst-case performance. If $lb \leq S^* \leq ub$, then lb (ub) is called **a** lower (upper) bound on the optimal solution. Clearly, lb (ub) should be as large (small) as possible, with the goal of having $lb = ub = S^*$. In this paper, we present an efficient lower bound algorithm that computes a reasonably tight lower bound on the length of an optimal schedule, and a new highest lower-bound first (HLBF) scheduling algorithm to provide a near optimal solution for the SPP scheduling problem.

The rest of the paper is organized as follows. In Section 2, the superscalar pipelined processor model and the task model are formalized. In Section 3, previous work is reviewed. We present our lower bound algorithm in Section 4 and our scheduling algorithm in Section 5. Simulation data is detailed in Section 6, and conclusions are drawn in Section 7.

2 Problem Statement

The SPP scheduling problem takes as input the processor configuration and the task to be executed on the processor. In this section, we will describe the superscalar pipelined processor model, the task system, and the constraints on an SPP scheduling problem.

The time (number of cycles) required for executing an instruction in a pipeline is called the latency of the pipeline (instruction). If each stage takes one time unit, then the latency equals

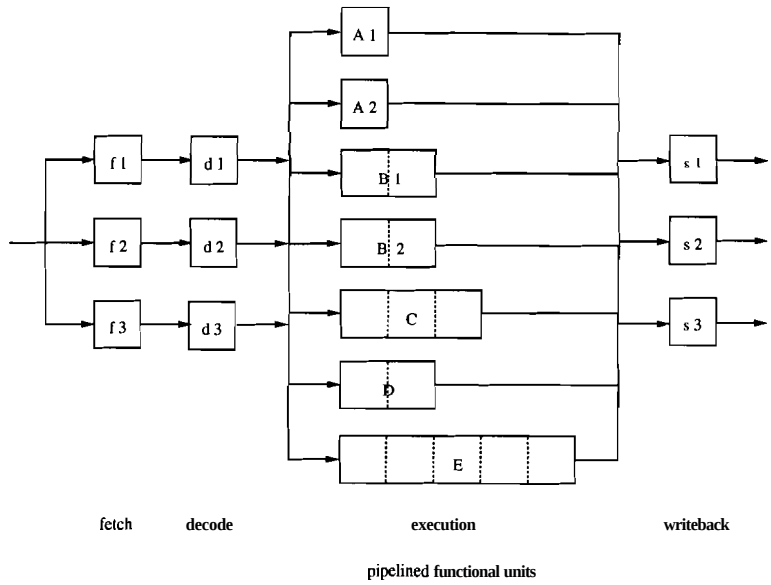


Figure 1: A superscalar pipelined processor with three fetch and decode unit:;, three writeback units, and seven pipelined functional units.

the number of stages in a pipeline. The number of instructions that can be issued (fetched and decoded) per cycle, M, is called the instruction issue rate. Note that a scalar pipelined processor [6, 7, 8, 22, 28, 30] is a superscalar pipelined processor with an instruction issue rate of one. It is assumed that the functional units are pipelined with a possibly different number of stages (some authors refer this architecture as superpipelining [3, 23]), so that a faster clock rate is possible. A superscalar pipelined processor with three fetch and decode units, three writeback units, and seven pipelined functional units is shown in Figure 1.

Let OP = (1,. . ., N_{op}} be the set of operation types. Each operation type k has two associated quantities: L_k is the latency, and m_k is the number of type-k pipelines. For example, the parameters in Table 1 represent the superscalar pipelined processor in Figure 1. We assume that the functional units are fully pipelined (i.e., one instruction can be issued per cycle in each pipeline).

A set of instructions (or a task) I = (1,. . . , n) is to be scheduled on the superscalar pipelined processor. Each instruction is associated with an operation type. Let t_i be the time required for executing instruction i in a pipeline (of the same type). A partial order $\prec$ specifies the precedence relation between instructions. If i $\prec$ j and instruction i is issued at time t, then the earliest time that instruction j can be issed is $t + t_i$.

A task system can be represented by a directed graph (called a task *graph*), G.,in which vertices represent instructions and arcs represent precedence relations. It is assumed that the task graph is

operation type	*latency*	*#(pipelines)*
A	*1*	*2*
B	*2*	*2*
C	*3*	*1*
D	*2*	*1*
E	*5*	*1*

Table 1: A set of parameters for the superscalar pipelined processor in Figure 1. Note $L_A = 1$ and $m_A = 2$.

acyclic [27] because scheduling is done within a basic block or a trace [16, 19] (loop unrolling can be done before scheduling). If times are associated with the vertices, then the cost of a path P $(\sum_{i \in P} t_i)$ becomes the total time required to complete **all** instructions on the path. If there is an arc from i to j in G, then i is called a parent of j and j is called a child of i. If there is a path from i to j in G, then i is called an ancestor of j and j is called a descendant of i. The set of ancestors of i is denoted A;; the set of descendants of i is denoted D;. A vertex i is called it head vertex if A_i is empty, a tail vertex if D; is empty. The set of head vertices of G is denoted $head(G)$; the set of tail vertices of G is denoted $tail(G)$. A subgraph of G with vertex set V is denoted as $G(V)$. For convenience, we will add two pseudo-vertices, 0 and X , with zero execution time to G, and add an arc from 0 to i if i is a head vertex, add an arc from i to X if i is a tail vertex. Thus, G becomes a single-entry, single-exit DAG.

There are two types of constraints in an SPP scheduling problem:

- Precedence Constraint: If instruction j depends on instruction i, then j cannot be issued until i has completed execution. The precedence constraint requires that an instruction cannot be issued until **all** of its parents (and thus ancestors) have completed execution.
- Capacity Constraints:
  - Fetching Unit: at most M instructions can be issued in each cycle.
  - Functional Unit: at most m_k type-k instructions can be issued in each cycle.

A *schedule* is a set of tuples $\{(s_i, p_i) : 1 \leq i \leq n)$, where s; is the time to issue instruction i, and p_i is the pipeline for executing instruction i. A feasible schedule is one that satisfies both

the precedence and capacity constraints. The length $|S|$ of a schedule S (starting at $t = 0$) is the maximal completion time over all instructions, i.e.,

$$|S| \equiv max\{s_i + t_i : 1 \leq i \leq n\} \quad (1)$$

An optimal schedule is a feasible schedule with minimum length.

3 Related Work

The scalar pipelined processor scheduling problem has been studied extensively [6, 7, 8, 22, 28, 30], but the superscalar pipelined processor scheduling problem has gained more attention in recent years [9, 10, 17, 25, 34]. Problems considered in the literature often assume uniform execution time for each instruction, which may not be a reasonable assumption since floating-point operations require more cycles for execution than fixed-point operations. For scalar pipeline scheduling, if the task graph is a tree or each pipeline contains at most two stages, then optimal solutions can be obtained [E, 28]; otherwise, the problem is NP-complete.

The SPP scheduling problem is closely related to the microcode compaction problem [15, 19, 34]. Davidson et al. [15] examined the performance of various compaction algorithms (first-come-first-serve, list scheduling, branch-and-bound algorithm, and critical path algorithm) that combine microoperations into microinstructions within a basic block. Shiau and Chung [34] apply these algorithms to superscalar pipeline scheduling problems with unit execution time instructions.

Fisher and Ellis developed a VLIW (Very Long Instruction Word) processor and a compiler to support it (16,19]. Note that the VLIW processor is roughly equivalent to a superscalar pipelined processor, where the instruction issue rate equals the number of pipelines. Fisher uses a ***trace*** scheduling technique to exploit the parallelism in programs [19]. A trace can be considered to be a single very large basic block [16].

Obviously, the complexity and cost of hardware depend on the instruction issue rate and the number of functional units. Furthermore, the maximal speed-up may not be achieved when the hardware becomes more complex. Questions relevant to designing a superscalar or VLIW processor include:

- What is the optimal instruction issue rate (or the word length) ?
- How many functional units are required for each type of operation ?

Clearly, the instruction issue rate must be less than or equal to the total number of pipelines. Butler et al. [9] suggested that 2.0 to 5.8 instructions per cycle is sustainable if the hardware is properly balanced. In our simulations (see Figure 8), if a task matches the hardware or the type of instructior s is uniformly distributed, then issuing five instructions per cycle can achieve a speed-up; however, if the task is a bad match with the hardware, then issuing more than three instructions per cycle does not provide any speed-up.

4 Lower Bounds for the SPP Scheduling Problem

Two obvious lower bounds for the SPP scheduling problem, similar to those in [1, 18, 24, 27], can be obtained as follows:

- Critical Path: Let h_X be 0, define the height h; of a vertex i as:

$$\text{h;} := \max\{h_j : j \in child(i)\} + t_i \tag{2}$$

  Because instructions on any path must be executed sequentially, the cost of any path is a lower bound of S^*. Hence, $h_{max} \equiv \max\{h_i : 1 \leq \text{i} \leq \text{n})$ is a lower bound of S^*.

- Fetching Capacity Constraint: If there are n instructions and the instruction issue rate is M, then $\lceil n/M \rceil$ is a lower bound of S^*.

A preliminary lower bound is $LB_1 := max\{h_{max}, \lceil n/M \rceil\}$. Although LB_1 provides a good estimate of S^* for small M, the error increases significantly when M increases and when the architecture does not *match* the task as we will show in Section 6.

4.1 A Tighter Lower Bound

In this section, we introduce various labels and co-labels (see Table 2) to compute a tighter lower bound for the SPP scheduling problem.

A label (height, density, lower-bound) is computed over the descendant set; a co-label (co-height, co-density, co-lower-bound) is computed over the ancestor set. The height h_i is computed as in Equation 2. The density d_i is obtained by considering the functional unit capacity constraint, i.e., at most m_k type-k instructions can be issued per cycle. The lower-bound lb_i is computed by considering; the height and the density. A counterpart of h_i, d_i, and lb_i can be computed similarly over the ancestor set. The labels and co-labels are summarized in Table 2.

label	notation	definition
height	h_i	$\max\{h_j : \mathrm{j} \in child(i)\} + t_i$
co-height	h'_i	$\max\{h'_j : j \in parent(i)\} + t_i$
density	d_i	$\max_{D_i} \begin{cases} \lceil \frac{N}{M} \rceil + t_{min} - 1, \\ \max\{\lceil \frac{n_k}{m_k} \rceil + L_k - 1 : 1 \le k \le N_{op}\} \end{cases}$
co-density	d'_i	$\max_{A_i} \begin{cases} \lceil \frac{N}{M} \rceil + t_{min} - 1, \\ \max\{\lceil \frac{n_k}{m_k} \rceil + L_k - 1 : 1 \le k \le N_{op}\} \end{cases}$
lower-bound	lb_i	$\max \begin{cases} h_i, \\ d_i + t_i, \\ \max\{lb_j : j \in child(i)\} + t_i \end{cases}$
co-lower-bound	lb'_i	$\max \begin{cases} h'_i, \\ d'_i + t_i, \\ \max\{lb'_j : j \in parent(i)\} + t_i \end{cases}$

Table 2: Definitions of height h_i, density d_i, lower-bound lb_i, and their co-label counterparts.

Lemma 1 describes the way we partition a problem into subproblems to determine a tighter lower bound.

Lemma 1 (Partition) If A; is the set of ancestors of i and D_i is the set of descendants of i, then $S^*(G(A_i + i + \mathrm{D};)) = S^*(G(A_i)) + t_i + S^*(G(D_i))$.

Proof: It follows from the fact that i cannot be issued until all ancestors of i have completed execution, and no descendants of i can be issued until i has completed execution. ■

We next present two lemmas that are used by Theorem 1 which defines d;. We then present Theorem 2 which defines lb_i. Instruction i is called a last-issued instruction if $\forall j, s_i \ge s_j$. Note that in any feasible schedule, the instructions issued in the first cycle must be head vertices and the last-issued instructions must be tail vertices.

Lemma 2 For any subgraph G' of G, let $t_{min} = \min\{t_i : \mathrm{i} \in tail(G')\}$. If there are N vertices in G' and the instruction issue rate is M, then $S^*(G') \ge \lceil \frac{N}{M} \rceil + t_{min} - 1$.

Proof: Let i be a last-issued instruction in an optimal schedule. Suppose i is issued at time t,

then t $\geq \lceil \frac{N}{M} \rceil - 1$. Instruction i must be a tail vertex, hence t; $\geq t_{min}$. Instruction i cannot be completed before $t + t_i \geq \lceil \frac{N}{M} \rceil - 1 + t_{min}$. ■

Lemma 3 For any subgraph G' of G, if there are n_k type-k instructions in G', then

$$S^*(G') \geq \lceil \frac{n_k}{m_k} \rceil + L_k - 1$$

where L_k is the latency of type-k instructions.

Proof: At least one of the type-k instructions must be issued at time t $\geq \lceil \frac{n_k}{m_k} \rceil - 1$. This instruction cannot be completed before t $+ L_k \geq \lceil \frac{n_k}{m_k} \rceil - 1 + L_k$. ■

Theorem 1 Let D; be the set of descendants of vertex i. Let $t_{min} = \min\{t_i : \text{i} \in tail(G(D_i))\}$, n_k be the number of type-k instructions in D_i, and $|D_i| = \text{N}$. Define the density of vertex i:

$$\text{di} = \max \begin{cases} \lceil \frac{N}{M} \rceil + t_{min} - 1, \\ \max\{\lceil \frac{n_k}{m_k} \rceil + L_k - 1 : 1 \leq k \leq N_{op}\} \end{cases}$$

Then $S^*(G(D_i)) \geq$ d;. It follows that $S^*(G(D_i)) \geq d_{max} = \max\{d_i : 0 \leq \text{i} \leq \text{n}\}$.

Proof: It follows from Lemmas 2 and **3** because G(D_i) is a subgraph of G. ■

Theorem 2 If lb_X=0 (note that X is the pseudo vertex added to G to make it single-exit), and we define the lower-bound lb_i of a vertex i as:

$$lb_i = \max \begin{cases} h_i, \\ d_i + t_i, \\ \max\{lb_j : j \in child(i)\} + t_i \end{cases}$$

Then $S^*(G(D_i + i)) \geq lb_i$.

Proof: It can be proven by induction on depth.

(i) basis: $S^*(X) \geq lb_X$.

(ii) hypothesis: suppose $S^*(G(D_j + \text{j})) \geq lb_j$.

(iii) induction: Let i be a parent of j. $S^*(G(D_i)) \geq$ d; by Theorem 1. $S^*(G(D_i)) \geq S^*(G(D_j+j)) \geq lb_j$ because $G(D_j + \text{j})$ is a subgraph of $G(D_i)$. By Lemma 1, $S^*(G(D_i + \text{i})) = S^*(G(D_i)) + t_i$. It follows that $S^*(G(D_i + \text{i})) \geq d_i + t_i$ and $S^*(G(D_i + \text{i})) \geq lb_j + t_i$. h_i is the length of the longest path from i to X. Hence, $S^*(G(D_i + \text{i})) \geq h_i$. The conclusion follows directly. ■

LB_2(G,M)

1. compute the density d_i and co-density d'_i for each vertex
2. compute the height h_i and co-height h'_i for each vertex
3. compute the lower-bound lb_i and co-lower-bound lb'_i for each vertex
4. return $\max\{lb_i - \text{ti} + lb'_i : 0 \leq i \leq X\}$

Figure 2: LB_2, a lower bound algorithm for an SPP scheduling problem.

The duals of Theorems 1 and 2 for co-labels are parallel to the previous proofs.

An algorithm LB_2 for computing a tight lower bound for an SPP scheduling problem is shown in Figure 2. To compute the density and co-density requires finding the transitive closure of G [2, 4, 14] which can be done in $O(n^3)$ time. The other labels (h_i, h'_i, lb_i and lb'_i) can be computed in a depth-first fashion in $O(n + |G|)$ time, where n is the number of vertices and $|G|$ is the number of arcs in G. Hence, the overall time complexity is $O(n^3)$, which is dominated by the time to compute the transitive closure of G. Theorem 3 demonstrates that LB_2 computes a lower bound for an SPP scheduling problem.

Theorem 3 $S^* \geq LB_2 = \max\{lb_i + lb'_i - t; : 0 \leq i \leq X)$.

Proof: For each vertex i, $S^*(G(D_i)) \geq lb_i - t;$ by Theorem 2. Similarly, $S^*(G(A_i)) \geq lb'_i - t_i$. Hence, by Lemma 1,

$$S^*(G(A_i + i + D;)) = S^*(G(A_i)) + t_i + S^*(G(D_i)) \geq (lb_i - t_i) + t; + (lb'_i - t_i) = lb_i + lb'_i - t;$$

It follows that $S^* \geq \max\{S^*(G(A_i + i + D;)) : 0 \leq i \leq X) \geq LB_2$. ■

5 Highest Lower-Bound First Algorithm

In this section, we present a heuristic algorithm for the SPP scheduling problem. List scheduling heuristics have been used extensively by many researchers for scheduling problems [12, 24, 27]. A list scheduling algorithm assigns each vertex a label, forms a priority queue of the vertices in non-increasing (or non-decreasing) order by label, and then schedules vertices in the order on the list. Adam et al. discussed several list scheduling heuristics in [1]. As lb_i is a good lower bound on the length of an optimal schedule for vertex i and its descendants, it should serve as a powerful heuristic

```
HLBF(G,M)
1. T := 0
2. let Q be the set of unscheduled available tasks at time T
3. if Q is empty, then return
4. m := 0, n_k := 0
5. while m < M and Q is not empty
6.     retrieve the instruction i in Q with highest priority lb_i (assume i is of type k)
7.     if i is executable at time T, then
8.         schedule i at time T, rn := rn + 1, n_k := n_k + 1
9.     end
10. end
11. T := T + 1, goto step 2
```

Figure 3: A highest lower-bound first scheduler, where M is the instruction issue rate.

for scheduling. A highest lower-bound first algorithm (HLBF) is shown in Figure 3. The lower-bound lb_i for each vertex is computed before scheduling. We say that an instruction is available at time T if all of its parents have been scheduled at a time earlier than T. An available instruction i (of type-k) is executable at time T if the number of instructions scheduled at time T is less than M, the number of type-k instructions scheduled at time T is less than m_k, and for each parent j of i, $s_j + t_j \leq$ T. In cycle T, an available instruction with highest priority (lb_i) is selected. If it is executable, it is scheduled at time T, otherwise the next available instruction is considered. If all available instructions are examined or the number of instructions scheduled at time T equals M, then T is increased and the process continues until all instructions are scheduled.

6 Simulation Analysis

To test the effectiveness of our lower bounds and the HLBF algorithm, we have simulated scheduling randomly generated DAGs on the superscalar pipelined processor shown in Table 1 with a vector p specifying the occurrence probability for each operation. For example, $p = (.47, .313, .169, .024, .024)$ indicates that the probability for an instruction to be of type A is 0.47, the probability for an instruction to be of type B is 0.313, etc. The type of each instruction is randomly generated based on the given probabilities.

```
DAG-GENERATOR(n,q)
1. if RANDOM() < 0.5, then A(1,2) := 1
2. for j = 3 to n
3.     r := RANDOM()
4.     d := 2
5.     if r < q0, then d := 0
6.     if q0 ≤ r < q0 + q1, then d := 1
7.     pick d numbers i ∈ [1, j − 1] and set A(i, j) := 1
8. end
9. randomly reorder the indices and modify A accordingly
```

Figure 4: A random DAG generator, where $A(i, \mathbf{j}) = 1 \iff \mathbf{i} \prec \mathbf{j}$, n is the number of instructions, and q is the precedence probability vector.

For RISC processors [26, 32], each instruction typically has at most two operands. Hence, we assume that each vertex has at most two parents. The partial order specifying the precedence relations is randomly generated based on a precedence probability vector q, where q_i is the probability that an instruction has i parents, i = 0, 1, 2. A random DAG generator is shown in Figure 4, for which n is the number of instructions, and q is the precedence probability vector. Step 9 is to renumber the indices to create more randomness. Obviously, DAG-GENERATOR randomly generates DAGs that allow at most two parents for each vertex.

Simulations were done on an IBM RS/6000 workstation, using RANDOM, the random number generator provided by UNIX, for n = 100...1000 with an increment of 100, M = 2...7 and q = (0.3, 0.4, 0.3). Ten random instances are scheduled for each (n, M)-pair. We consider three different sets of occurrence probabilities:

1. p = (.47, .313, .169, .024, .024), which represents a good match between the hardware and the task.
2. p = (.2, .2, .2, .2, .2), where **all** instructions have uniform occurrence probabilities.
3. p = (.169, .024, .024, .47, .313), which represents a poor match between the hardware and the task.

To provide an estimate of the actual error rate, we define the *approximate* error rate of an algorithm using lb as an optimal solution estimate as: $r(lb) := (\text{Solution} - lb)/lb$, where Solution is

the heuristic solution provided by the algorithm. Note that $r(lb)$ is an upper bound on the actual error rate. The distribution of $r(LB_2)$ (over all instances) of the HLBF algorithm for the three cases is shown in Figure 5. In Figures 6 and 7, the average $r(LB_1)$ and $r(LB_2)$ are depicted as a function of n and M. The heuristic solutions and lower bounds for n = 1000 are shown in Figure 8. The heuristic solutions and lower bounds for M = 5 are shown in Figure 9. The simulation results show that:

- LB_1 is an especially poor estimate of the optimal solution when the instruction issue rate increases, or when the hardware does not match well with the task (see case **3** of Figure 6).
- LB_2 is a much tighter lower bound than LB_1 (compare Figures 6 and 7). The average peak values of $r(LB_1)$ and $r(LB_2)$ for the three cases are listed in the following table:

	mean		max	
case	$r(LB_1)$	$r(LB_2)$	$r(LB_1)$	$r(LB_2)$
1	25.43%	5.5%	66.6%	20%
2	16.89%	1.71%	63.95	6.9%
3	109.8%	0.84%	237.41%	6.255%

- LB_2 provides a good estimate on the optimal solution in most cases (see Figure 5).
- Intuitively, increasing the instruction issue rate may decrease the overall execution time of a task. However, the speed-up may saturate when the instruction issue rate reaches a certain value (see Figure 8). For example, no speed-up can be achieved beyond M = 5 for cases 1 and 2, and M = **3** for case **3.** This result partially supports the previous conclusion made by Butler et al. in [9]. We call this saturation point the maximal parallelism of the problem instance. Increasing the instruction issue rate beyond this value increases the code size without reducing the code execution time, and hence, is not desirable. This is because the functional unit capacity constraint becomes the dominant component in the lower bound LB_2.
- The solutions are bounded from below by h_{max}, $\lceil n/m \rceil$ and d_{max} (see Figures 8 and 9). h_{max} usually remains constant as the number of instructions increases.
- The critical path length h_{max} seems to be an unimportant factor in the lower bound, as might be expected.

7 Conclusions

In this paper, we have considered the scheduling of a superscalar pipelined processor without hardware interlocks. This architecture has the advantages of combining the benefits of the VLIW and superscalar processors, while avoiding the drawbacks. A lower bound algorithm LB_2 computes a tight lower bound on the length of an optimal schedule. An efficient scheduling algorithm HLBF provides a good schedule for tasks to be executed on the superscalar pipelined processor such that the compiled code is free of pipeline hazards. The scheduling algorithm HLBF uses the lower bound computed by LB_2 as a heuristic for selecting instructions for scheduling. The simulation data show that lb_i is a powerful heuristic, and LB_2 is very close to the heuristic solution, which suggests that LB_2 is a good lower bound on the optimal solution. However, it is possible to obtain a tighter lower bound when the task matches the hardware.

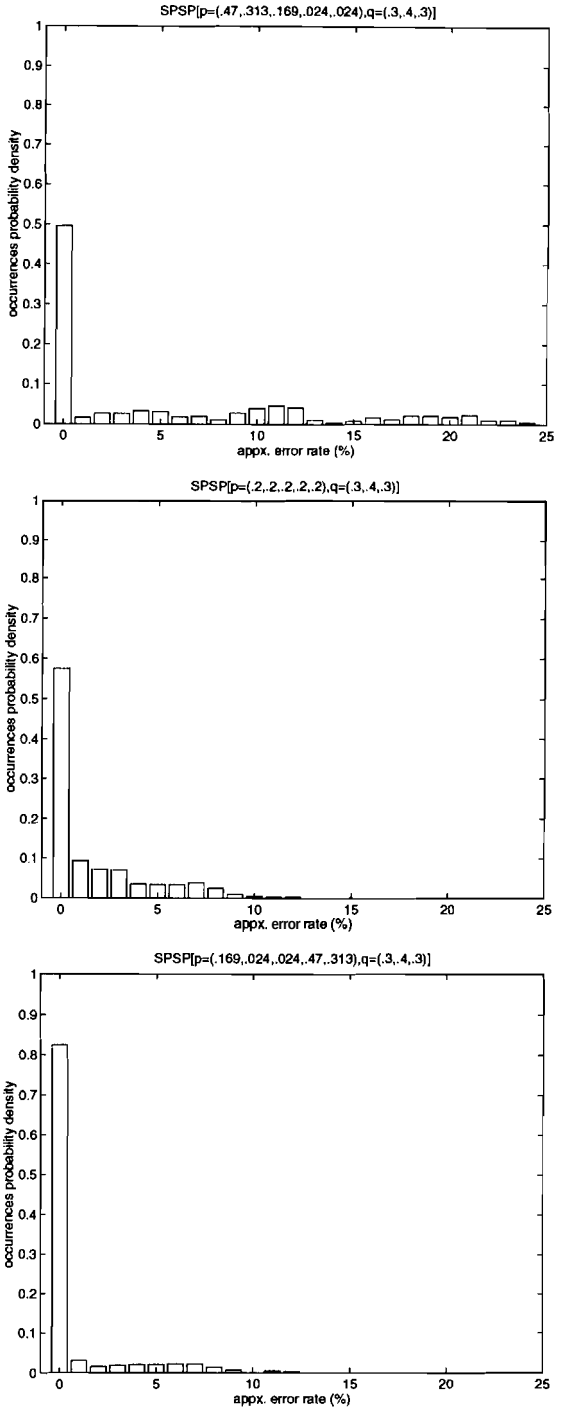


Figure 5: The distribution of the approximate error rate $r(LB_2)$ (over all instances) of the HLBF algorithm for the three sets of occurrence probabilities.

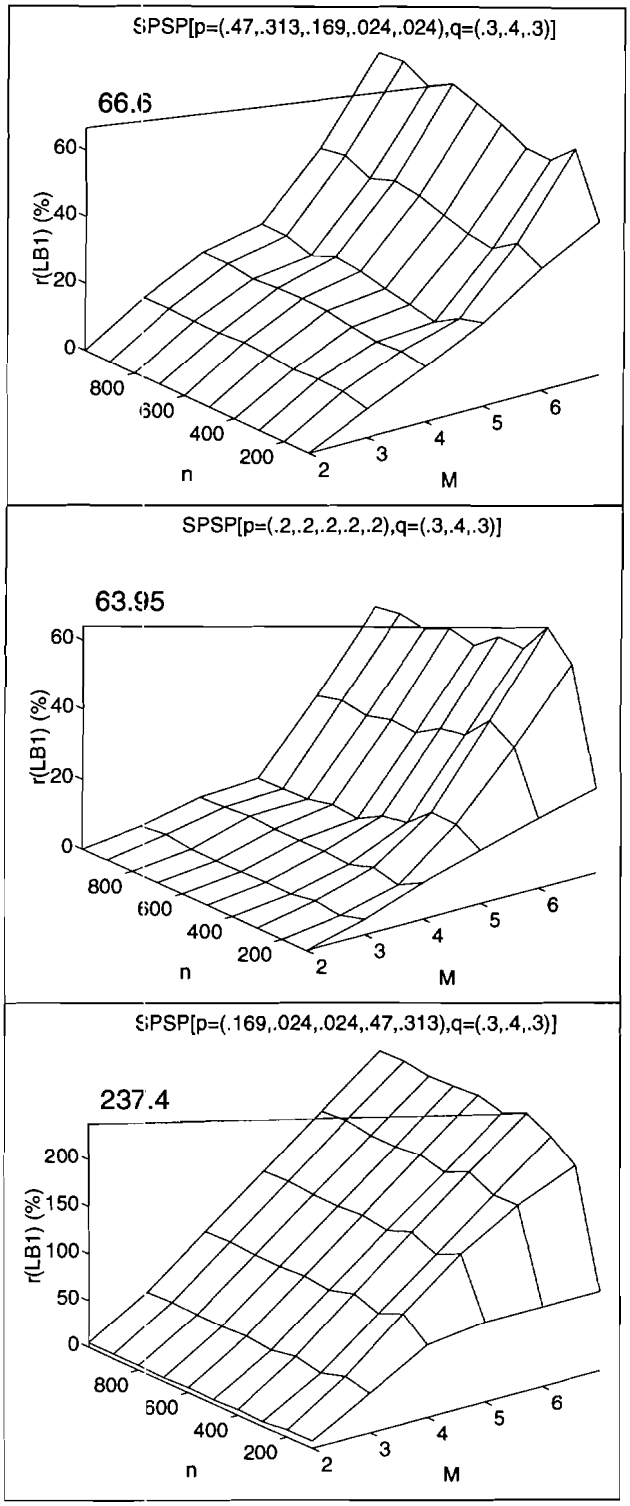


Figure 6: The average $r(LB_1)$ of the HLBF algorithm as a function of n and M.

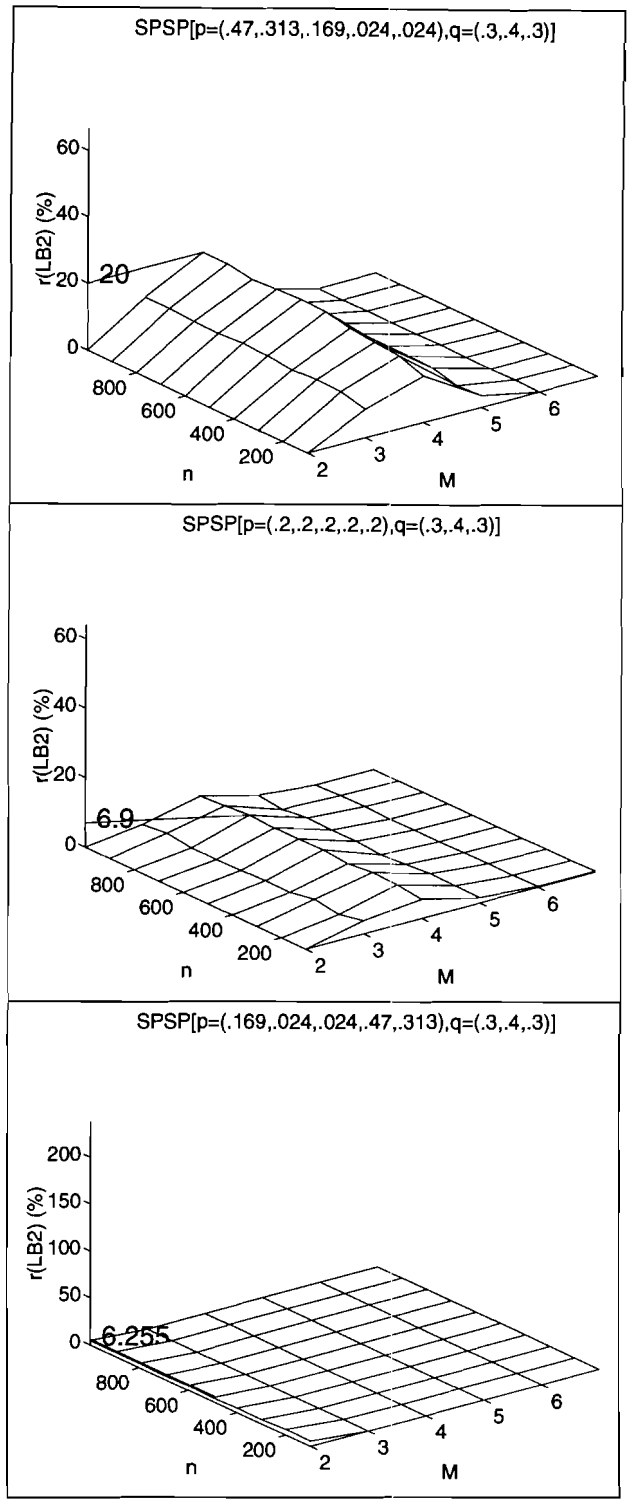
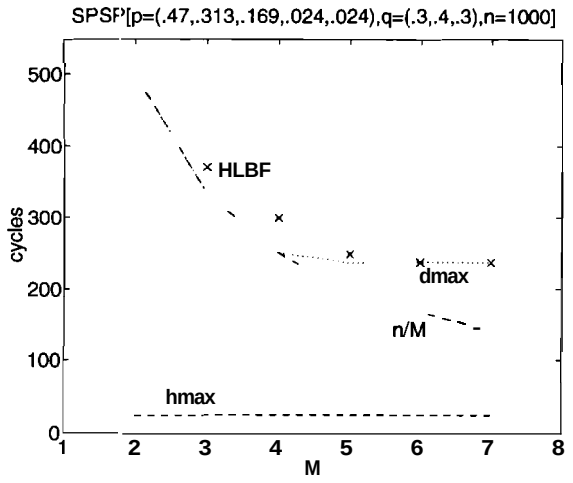
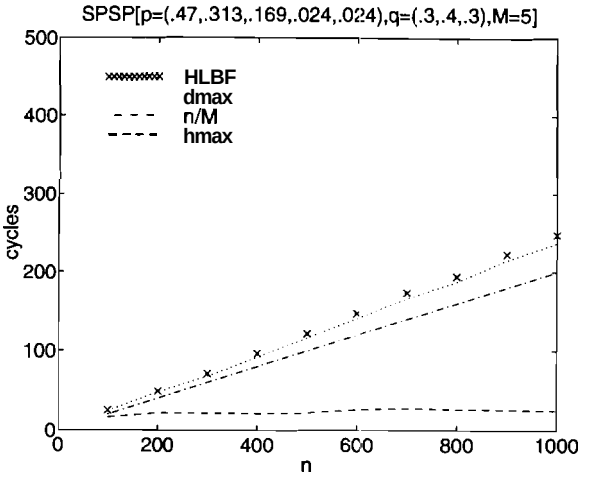
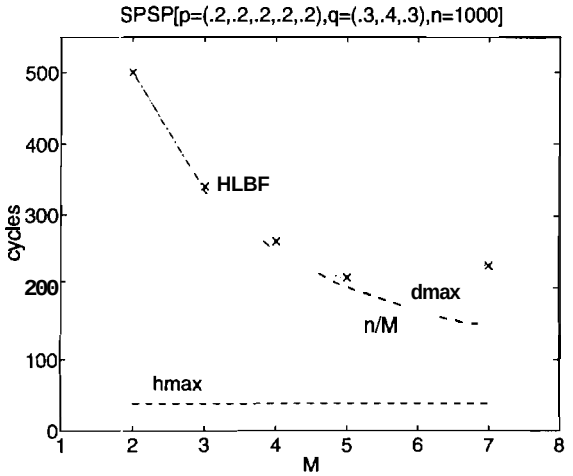
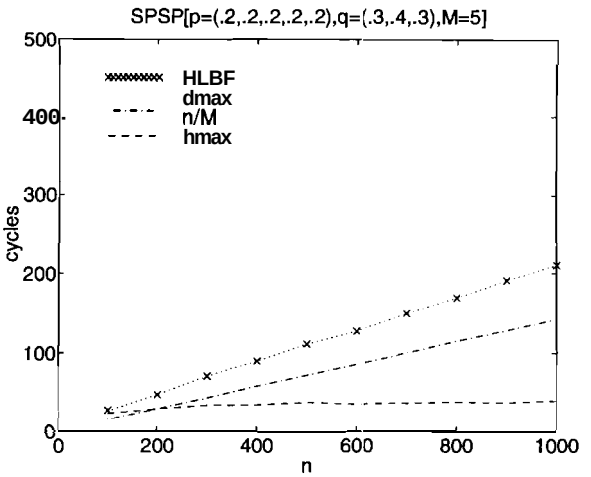


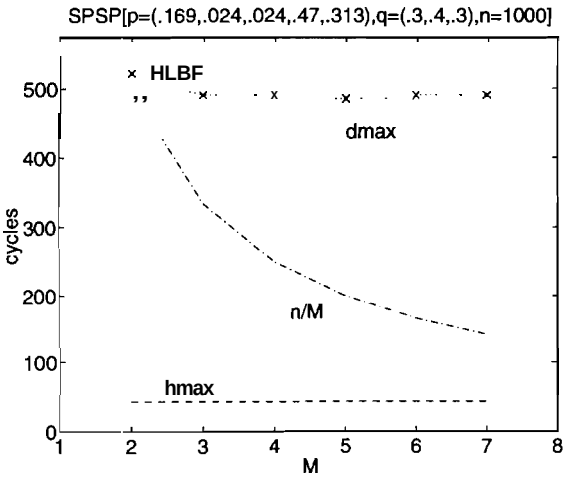
Figure 7: The average $r(LB_2)$ of the HLBF algorithm as a function of n and M.











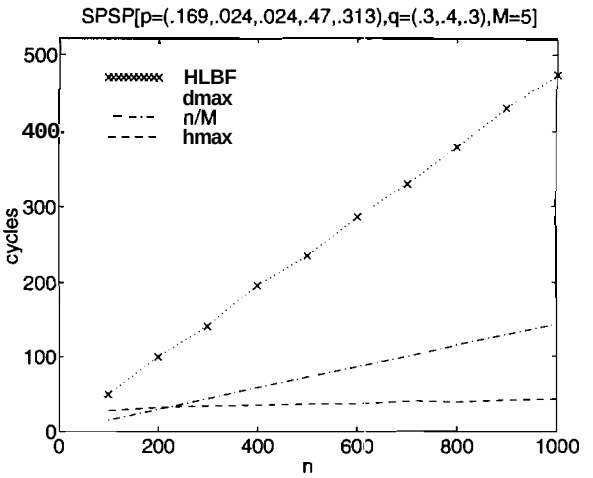


Figure 8: The heuristic solution and lower bounds for n=1000.

Figure 9: The heuristic solution and lower bounds for M=5.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. Communications of the ACM, 17(12):685–690, December 1974.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, San Francisco, CA, 1976.

[3] B. Asghar. A superpipeline approach to the MIPS architecture. In *COMPCON* '91, Spring, pages 8–12, 1991.

[4] S. Baase. Computer Algorithms. Addison-Wesley Publishing Company, San Diego, CA, 1988.

[5] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye. The IBM RISC system/6000 processor: Hardware overview. IBM journal of research and *development*, 34(1):12–22, January 1990.

[6] W. Baxter and R. Arnold. Code restructuring for enhanced performance on a pipelined processor. In COMPCON '91, Spring, pages 252–260, 1991.

[7] D. Bernstein. An improved approximation algorithm for scheduling pipelined machines. In *International* Conference on Parallel Processing, pages 430–433, 1988.

[8] J. Bruno, J. W. Jones, and K. So. Deterministic scheduling with pipelined processors. IEEE Transactions on Computers, C-29(4):308–316, April 1980.

[9] M. Butler, T. Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In 1991 IEEE 18th Annual *International* Symposium on *Computer* Architecture, pages 276–286, 1991.

[10] H. C. Chou and C. P. Chung. A bound analysis of scheduling instructions on pipelined processors with a maximal delay of one cycle. Parallel Computing, 18:393–399, 1992.

[11] J. Circello and F. Goodrich. The Motorola 68060 microprocessor. In *COMPCON* '93, Spring, pages 73–78, 1993.

[12] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta. Informatica*, 1:200–213, 1972.

[13] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. IEEE Transactions on *Computers*, 37(8):967–979, August 1988.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to *Algorithms*. McGraw-Hill Book Company, New York, NY, 1990.

[15] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett. Some experiments in local microcode compaction for horizontal machines. IEEE Transactions on *Computers*, C-30(7):478–477, July 1981.

[16] J. R. Ellis. Bulldog: *A Compiler* for VLIW Architectures. The MIT Press. Cambridge, Massachusetts, 1986.

[17] E. S. T. Fernandes and F. M. B. Barbosa. Effects of building blocks on the performance of superscalar architectures. In 1992 IEEE 19th Annual International *Symposium* on Computer Architecture, pages 36–45, 1992.

[18] E. B. Fernandez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules. IEEE Transactions on Computers, C-22(8):745–751, August 1973.

[19] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers, C-30(7):478–490, July 1981.

[20] J. A. Fisher. Very long instruction word architectures and the ELI-512. In The 10th *Annual International* Symposium on Computer Architecture, pages 140–150, June 1983.

[21] M. R. Garey and D. S. Johnson. Computers and *Intractability*. W.H. freeman and Company, San Francisco, CA, 1979.

[22] T. R. Gross. Code optimization techniques for pipelined architectures. In. COMPCON '83, *Spring*, pages 278–285, 1983.

[23] J. L. Hennessy and D. A. Patterson. Computer Architecture - A Quantitative Approach. Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[24] T. C. Hu. Parallel sequencing and assembly line problems. Oper. Res., 9:841–848, November 1961.

[25] W. M. W. Hwu and P. P. Chang. Exploiting parallel microprocessor microarchitectures with a compiler code generator. In *1988* IEEE Symposium on Computer *Architecture*, pages 45–53, 1988.

[26] G. Kane and J. Heinrich. MIPS RISC Architecture. Prentice Hall, Englewood Cliffs, NJ, 1992.

[27] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Transactions on Computers, C-33(11):1023–1029, November 1984.

[28] H. F. Li. Scheduling trees in parallel/pipelined processing environments. IEEE Transactions on Computers, C-26(11):1101–1112, Nov. 1977.

[29] C. R. Moore. The PowerPC 601 microprocessor. In COMPCON *'93,* Spring, pages 73–78, 1993.

[30] A. Nisar. Optimal code scheduling for multiple pipeline processors. Master's thesis, Purdue University, 1990.

[31] R. R. Oehler and R. D. Groves. IBM RISC system/6000 processor architecture. IBM journal of research and development, 34(1):23–36, January 1990.

[32] D. A. Patterson. Reduced instructions set computers. Communications of the ACM, 28(1), January 1985.

[33] A. Saini. An overview of the Intel Pentium processor. In COMPCON *'93,* Spring, pages 60–62, 1993.

[34] Y. H. Shiau and C. P. Chung. Adoptability and effectiveness of microcode compaction algorithms in superscalar processing. Parallel Computing, 18(5):497–510, 1992.